RNN Language Modeling (revisited)

CS 780/880 Natural Language Processing Lecture 18

Samuel Carton, University of New Hampshire

Last lecture

RNNs for language modeling

Generating text

- Greedy decoding
- Random sampling
- Beam search decoding

Training RNNs

- Teacher forcing
    - Exposure bias
- Alternatives
    - Minimum risk, reinforcement learning, GANs

Last lecture

RNNs for language modeling

Generating text

- **Greedy decoding**
- **Random sampling**
- Beam search decoding

Training RNNs

- **Teacher forcing**
    - Exposure bias
- Alternatives
    - Minimum risk, reinforcement learning, GANs

Review: Language modeling

Basic idea: Given words $\{w^0, w^1, w^2, \ldots, w^{t-1}\}$, we want to be able to reliably predict w^t

If we can do this, we can:

- Generate new text
- Assess the overall likelihood of a piece of text
- (In 2023) talk to the model like it is a person and make it do stuff for us
    - Prompt engineering

Lecture content borrowed from https://courses.engr.illinois.edu/cs447/fa2020/index.html

Word logits

sparse unigram word logits$_0$

Output layer

hidden vector$_{-1}$

hidden vector$_0$

state vector$_{-1}$

LSTM cell

state vector$_0$

dense word vector$_0$

Pretrained word embeddings (frozen)

sparse unigram word vector$_0$

sparse unigram word logits$_1$

Output layer

hidden vector$_1$

LSTM cell

state vector$_1$

dense word vector$_1$

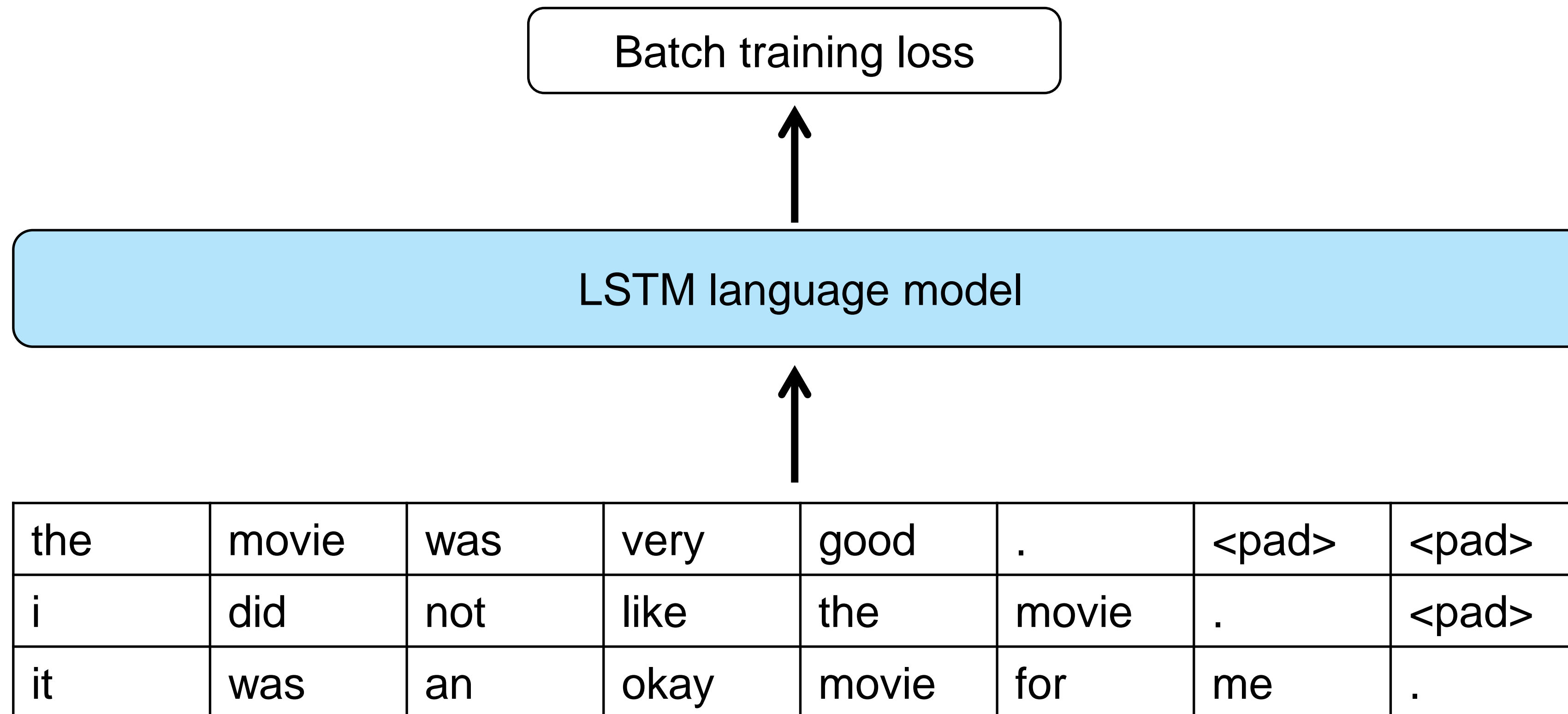
Pretrained word embeddings (frozen)

sparse unigram word vector$_1$

= input/output tensor

= model parameter

Batch training



Vector generation

as	good	as	a	movie

LSTM language model

the	movie	was

as	good	as	a	movie

Preliminaries

Usual preliminaries:

1. Load GloVE vectors using Gensim
   - Add special tokens: <pad>, <unk>, <sos>, <eos>
2. Load & preprocess SST-2 dataset
   - We'll be ignoring the label for this exercise
   - Includes:
     - Lower-casing, tokenization
     - Map to GloVE tokens
     - Add <sos> and <eos> tokens
3. Create dataset & dataloader
4. Install PyTorch Lightning

Preprocessing

```
1  display(dev_df)
```

	sentence	label	tokens	input_ids
0	it 's a charming and often affecting journey .	1	[<sos>, it, 's, a, charming, and, often, affecting, journey, ., <eos>]	[400002, 20, 9, 7, 12387, 5, 456, 7237, 3930, 2, 400003]
1	unflinchingly bleak and desperate	0	[<sos>, unflinchingly, bleak, and, desperate, <eos>]	[400002, 101035, 12566, 5, 5317, 400003]
2	allows us to hope that nolan is poised to embark a major career as a commercial yet inventive fi...	1	[<sos>, allows, us, to, hope, that, nolan, is, poised, to, embark, a, major, career, as, a, comm...	[400002, 2415, 95, 4, 824, 12, 13528, 14, 7490, 4, 17406, 7, 224, 432, 19, 7, 1196, 553, 24065, ...
3	the acting , costumes , music , cinematography and sound are all astounding given the production...	1	[<sos>, the, acting, ,, costumes, ,, music, ,, cinematography, and, sound, are, all, astounding,...	[400002, 0, 2050, 1, 10349, 1, 403, 1, 22181, 5, 1507, 32, 64, 23248, 454, 0, 618, 9, 24932, 278...
4	it 's slow -- very , very slow .	0	[<sos>, it, 's, slow, --, very, ,, very, slow, ., <eos>]	[400002, 20, 9, 2049, 65, 191, 1, 191, 2049, 2, 400003]
...	...	...	...	...
867	has all the depth of a wading pool .	0	[<sos>, has, all, the, depth, of, a, wading, pool, ., <eos>]	[400002, 31, 64, 0, 4735, 3, 7, 27989, 3216, 2, 400003]
868	a movie with a real anarchic flair .	1	[<sos>, a, movie, with, a, real, anarchic, flair, ., <eos>]	[400002, 7, 1005, 17, 7, 567, 41588, 17056, 2, 400003]
869	a subject like this should inspire reaction in its audience ; the pianist does not .	0	[<sos>, a, subject, like, this, should, inspire, reaction, in, its, audience, ;, the, pianist, d...	[400002, 7, 1698, 117, 37, 189, 11356, 2614, 6, 47, 2052, 89, 0, 9399, 260, 36, 2, 400003]
870	... is an arthritic attempt at directing by callie khouri .	0	[<sos>, ..., is, an, arthritic, attempt, at, directing, by, callie, khouri, ., <eos>]	[400002, 434, 14, 29, 57228, 1266, 22, 8044, 21, 63691, 79156, 2, 400003]
871	looking aristocratic , luminous yet careworn in jane hamilton 's exemplary costumes , rampling g...	1	[<sos>, looking, aristocratic, ,, luminous, yet, careworn, in, jane, hamilton, 's, exemplary, co...	[400002, 862, 21897, 1, 29085, 553, 203745, 6, 4917, 3959, 9, 21144, 10349, 1, 92361, 1829, 7, 8...

872 rows × 4 columns

Dataloader

```
1  torch.random.manual_seed(1234)
2  first_train_batch = next(iter(train_dataloader))
3  print('First training batch:')
4  print(first_train_batch)
5
6  print('First training batch sizes:')
7  print({key:value.shape for key, value in first_train_batch.items()})
```

```
First training batch:
{'input_ids': tensor([[400002,    307,     66,      3,  11114,   2720,      5,   5097,  31351,
         400003, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001],
        [400002,  42131, 400003, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001],
        [400002,     29,  51710,  37369,   2692,     12,   1144,   1003,     64,
            317,   2516,      2, 400003, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001],
        [400002,   2322, 400003, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001],
        [400002,  18519, 400003, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
         400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001, 400001,
```

Language model class

```
class LSTMLanguageModel(pl.LightningModule):
  def __init__(self,
               word_vectors:np.ndarray,
               vocab_size:int,
               learning_rate:float,
               padding_id:int,
               lstm_hidden_size:int=100, # how big the inner vectors of the LSTM will be,
               lstm_layers:int =2, # how many layers the LSTM will have
               dropout_prob:float=0.1,
               loss_print_interval=100,
               **kwargs):
    super().__init__( **kwargs)

    self.word_embeddings = torch.nn.Embedding.from_pretrained(embeddings=torch.tensor(word_vectors),
                                                               freeze=True)
    self.lstm = torch.nn.LSTM(input_size = word_vectors.shape[1], # The LSTM will be taking in word vectors
                              hidden_size = lstm_hidden_size,
                              num_layers=lstm_layers,
                              bidirectional=False, # We can't count on being able to proceed both backward and forward
                              dropout=dropout_prob,
                              batch_first=True # This is important. Set to False by default for some reason.
                              )

    # Output layer has to produce one logit per potential word, so the output size is vocab_size
    self.output_layer  = torch.nn.Linear(lstm_hidden_size, vocab_size)
    self.lstm_layers = lstm_layers
    self.learning_rate = learning_rate
    self.padding_id = padding_id # we'll need this later

    self.train_loss = MeanMetric()
    self.val_loss = MeanMetric()
    self.loss_print_interval = loss_print_interval
```

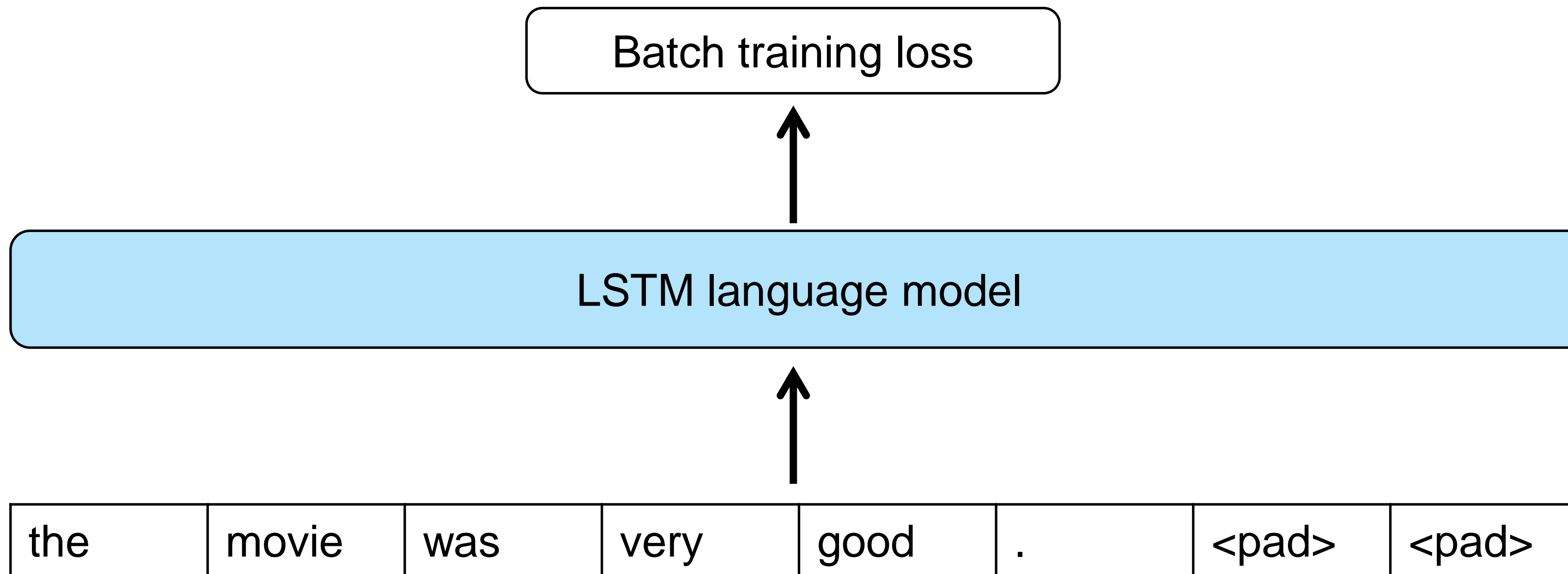
Batch training

Batch training loss

LSTM language model

the	movie	was	very	good	.	<pad>	<pad>
i	did	not	like	the	movie	.	<pad>
it	was	an	okay	movie	for	me	.

Batch training



Input IDs vs input tokens

I’m going to keep showing the actual tokens, but the reality is that the model is working with vectors/matrices of input IDs, that we find by doing lookups on the vector_model.

the	movie	was	very	good	.	<pad>	<pad>

=

3	14	7	58	138	6	400001	400001

Batch training (batch size = 1)

hiddens
(1 x sequence length x hidden size)

[0.03, 0.83, 0.09, …] → Output layer → [10.92, 0.15, -0.45, …]

logits
(1 x sequence length x vocab size)

LSTM

inputs_embeds
(1 x sequence length x embedding size)

[0.1, 0.04, 0.3, …]

Embedding layer

input_ids
(1 x seq length)

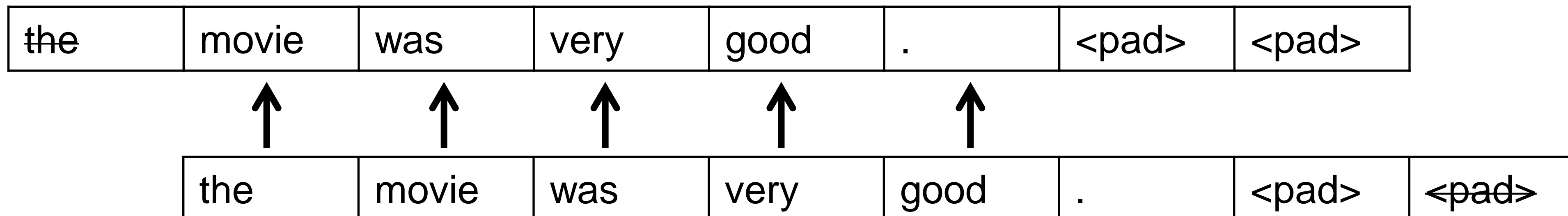
the	movie	was	very	good	.	<pad>	<pad>

Batch loss

To compute the loss for the batch, we'll compare the output logits to the input ID, shifted backward by 1

- So the output logits for "the" should be close to the unigram vector for "movie", etc.

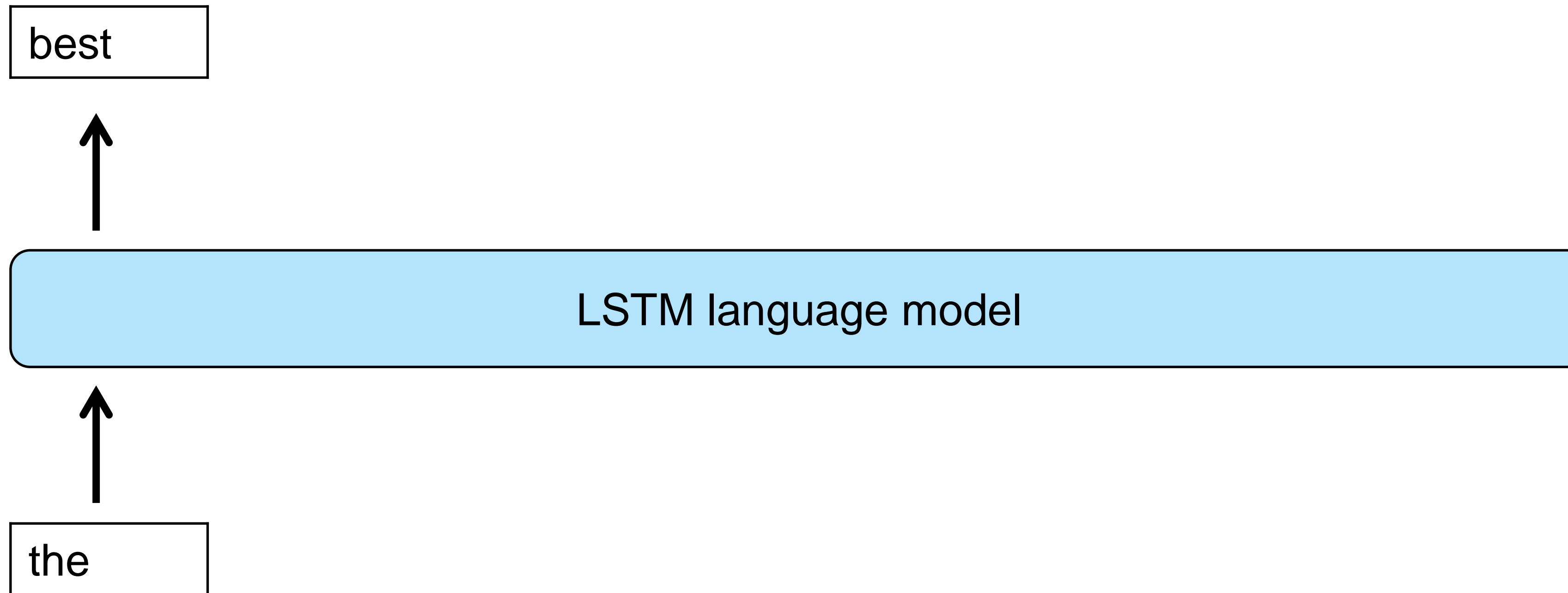
And we're still using cross-entropy loss (a.k.a negative log likelihood)



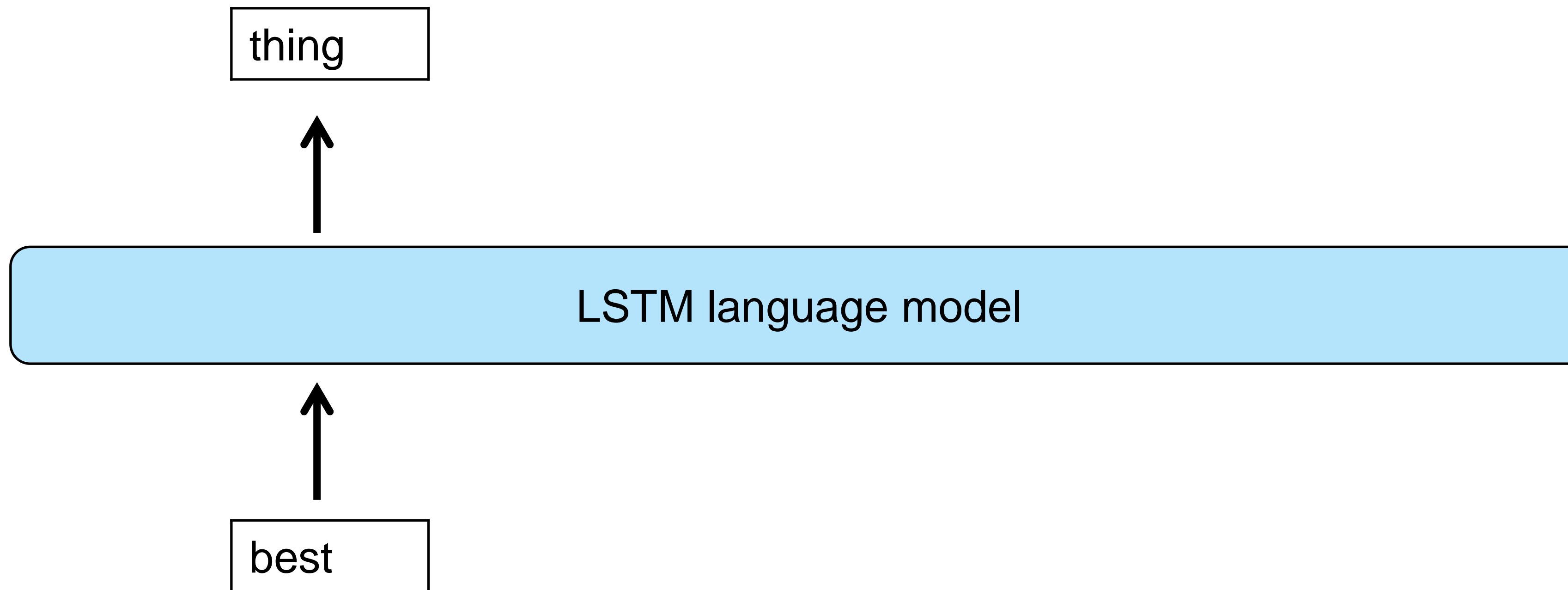
forward() method – training loss

```python
def forward(self,
            input_ids:torch.Tensor, #(batch size x max sequence length)
            ):

    # To do a training pass on the model, we're going to give it a batch as input, and
    # generate teacher-forcing loss based on that same batch
    inputs_embeds = self.word_embeddings(input_ids) #(batch size x max sequence length x embedding size)
    padding_mask = (input_ids != self.padding_id).int() #(batch size x max sequence length)
    input_lengths = padding_mask.sum(dim=1).detach().cpu() #(batch size)
    packed_embeddings = pack_padded_sequence(inputs_embeds, input_lengths, batch_first=True, enforce_sorted=False)
    packed_output, (final_hidden, final_state) = self.lstm.forward(packed_embeddings)

    hiddens, _ = pad_packed_sequence(packed_output, batch_first=True, padding_value=0.0, total_length=input_ids.shape[1])

    # The output logits here represent one (un-normalized) probability value for every possible word the model
    # could generate, for each input word
    output_logits = self.output_layer(hiddens) #(batch size x max sequence length x vocab size)


    # The target output for each input token t, is the true input token t+1
    # So we evaluate the output logits against the shifted-by-one version of the input tokens
    # And we don't impose a loss on the last input token
    # Pytorch cross entropy function still wants wants the vocab size to be the second dimension
    losses = torch.nn.functional.cross_entropy(output_logits[:,:-1].transpose(1,2), input_ids[:,1:], reduction='none')

    # Then the final thing we need to do is zero out the losses whenever the target token is a padding token
    padded_losses = losses * padding_mask[:,1:] # (batch size x max sequence length)
    loss = padded_losses.mean() #(1)

    # For the purpose of training, we're not going to bother sampling any actual text
    # We just generate the teacher-forcing loss
    return {'loss':loss}
```

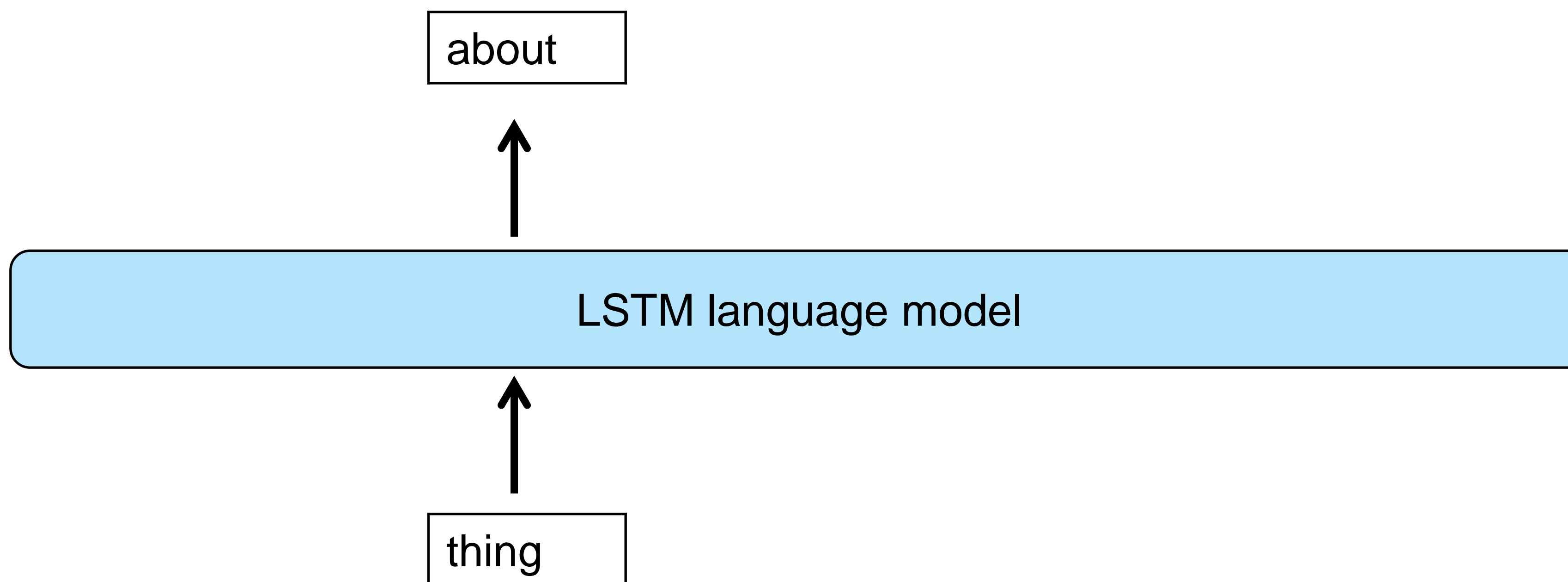
Vector generation



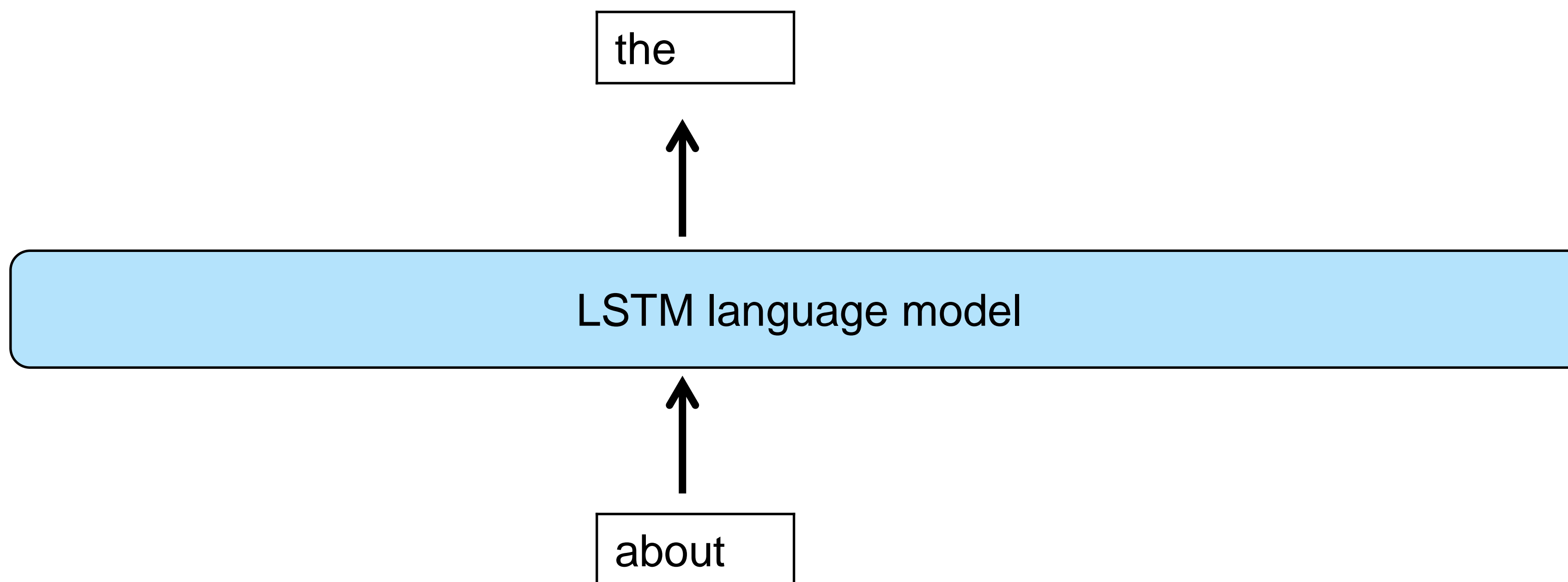
Vector generation



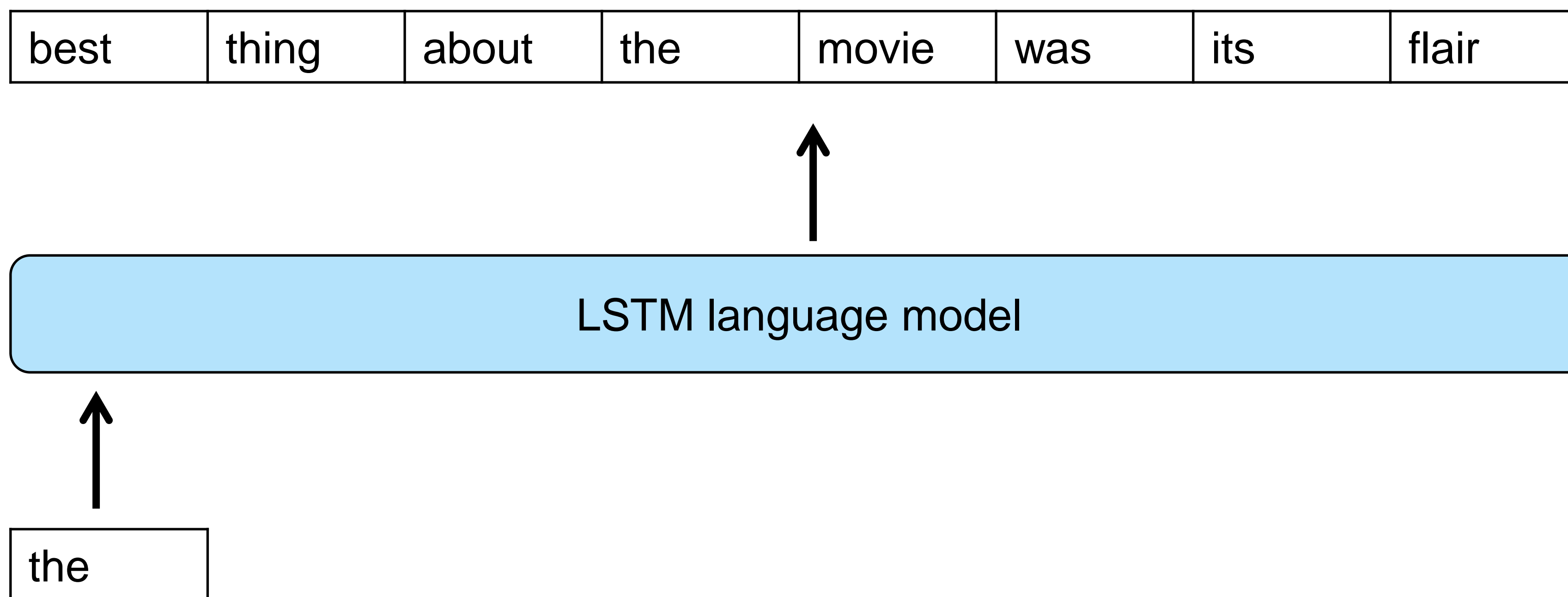
Vector generation



Vector generation



Vector generation



generate() method

```
def generate(self,
             input_ids:torch.Tensor, # Pain in the butt to do generation in batches, so assume this is 1-dimensional with
             #shape (1,sequence length) or (sequence length)
             output_length:int, # How many tokens to generate past the input sequence
             temperature:float=0.5, # How loosely to sample from the output distribution
             ):

  # If the input shape is (1, sequence length), make it (sequence length)
  if input_ids.ndim == 2: input_ids = input_ids.squeeze(1)

  # Remove padding tokens if they are present
  padding_mask = (input_ids != self.padding_id).int() #(batch size x max sequence length)
  input_length = padding_mask.sum().detach().cpu() #(batch size)
  input_ids = input_ids[0:input_length]
  inputs_embeds = self.word_embeddings(input_ids) #(sequence length x embedding size)

  # First we run the given sequence through the LSTM
  # Because we aren't using a batch of variable-length sequences, we don't have to bother with a packed padded sequence like above
  input_hiddens, (final_input_hidden, final_input_state) = self.lstm.forward(inputs_embeds) # (sequence length x lstm hidden size),
   #( (lstm layers x lstm hidden size), (lstm layers x lstm hidden size) )

  output_tokens = []
  # Then, for the rest of the desired output length, we generate one token at a time, conditioned on the previous generated token
  last_hidden, last_state = final_input_hidden, final_input_state
  last_logits = self.output_layer(last_hidden[-1])
  last_token_id = self.sample_token_id_from_logits(last_logits, temperature)
  output_tokens.append(last_token_id)
  for i in range(input_length, output_length):
    last_embeds = self.word_embeddings(last_token_id).unsqueeze(0)
    last_output, (last_hidden, last_state) = self.lstm.forward(last_embeds, (last_hidden, final_input_state))
    last_logits = self.output_layer(last_hidden[-1])
    last_token_id = self.sample_token_id_from_logits(last_logits, temperature)
    output_tokens.append(last_token_id)
    # break

  output_ids = torch.stack(output_tokens)

  return {'output_ids':output_ids,}
```

sample_token_id_from_logits() method

```python
def sample_token_id_from_logits(self,
                                logits:torch.Tensor, # (vocab size)
                                temperature: float
                                ):
  '''
  Take a logits vector and sample from it according to the specified temperature (higher = more random)
  '''

  if temperature == 0: # If temp is 0, select the index of the max logit
    token_id = logits.argmax()
  else: # Otherwise, divide all the logits by the temperature, then take a softmax to create a probability distribution
    # So the higher temperature is, the more the probabilities will be "smoothed out"
    probs = torch.softmax(logits / temperature, dim=0)
    token_id = torch.multinomial(probs, num_samples=1).squeeze(0)
  return token_id
```

evaluate() method

```
def evaluate(self, input_ids:torch.Tensor):
    '''
    Evaluate the log-likelihood of a given sequence under the model
    '''
    # If the input shape is (1, sequence length), make it (sequence length)
    if input_ids.ndim == 2: input_ids = input_ids.squeeze(1)

    # Remove padding tokens if they are present
    padding_mask = (input_ids != self.padding_id).int() #(batch size x max sequence length)
    input_length = padding_mask.sum().detach().cpu() #(batch size)
    input_ids = input_ids[0:input_length]
    inputs_embeds = self.word_embeddings(input_ids) #(sequence length x embedding size)
    input_hiddens, (final_input_hidden, final_input_state) = self.lstm.forward(inputs_embeds)
    # (sequence length x lstm hidden size), ( (lstm layers x lstm hidden size), (lstm layers x lstm hidden size) )

    output_logits = self.output_layer(input_hiddens)
    loss = torch.nn.functional.cross_entropy(output_logits[:-1], input_ids[1:])
    return {'negative_log_likelihood':loss}
```

PyTorch hooks

```python
# And then everything else is the same!
def configure_optimizers(self):
    return [torch.optim.Adam(self.parameters(), lr=self.learning_rate)]

def training_step(self, batch, batch_idx):
    result = self.forward(**batch)
    loss = result['loss']
    self.log('train_loss', result['loss'])
    if batch_idx % self.loss_print_interval ==0:
        print(f'Mean training loss (steps {batch_idx-self.loss_print_interval}-{batch_idx}): {self.train_loss.compute():.3f}')
        self.train_loss.reset()
    else:
        self.train_loss.update(loss.detach())
    # self.train_accuracy.update(result['py'], batch['tag_ids'])
    return loss

# def training_epoch_end(self, outs):
#     print(f'Epoch {self.current_epoch} training accuracy:', self.train_accuracy.compute())
#     self.train_accuracy.reset()

def validation_step(self, batch, batch_idx):
    result = self.forward(**batch)
    self.log('val_loss', result['loss'])
    self.val_loss.update(result['loss'])
    # self.val_accuracy.update(result['py'], batch['tag_ids'])
    return result['loss']

def validation_epoch_end(self, outs):
    print(f'Epoch {self.current_epoch} step {self.global_step} validation loss:', self.val_loss.compute())
    self.val_loss.reset()
```

Model training

```python
language_model = LSTMLanguageModel(word_vectors=vector_model.vectors,
                                   vocab_size = vector_model.vectors.shape[0],
                                   learning_rate = 0.001,
                                   padding_id = vector_model.key_to_index['<pad>'],
                                   lstm_hidden_size=100,
                                   lstm_layers=2,
                                   dropout_prob=0.1)

from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar

trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=3,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.1,
    )
trainer.fit(model=language_model,
            train_dataloaders=train_dataloader,
            val_dataloaders=dev_dataloader)
```

```
Mean training loss (steps -100-0): nan
Mean training loss (steps 0-100): 3.197
Mean training loss (steps 100-200): 2.581
Mean training loss (steps 200-300): 2.602
Epoch 0 step 336 validation loss: tensor(4.0279, device='cuda:0')
Mean training loss (steps 300-400): 2.483
Mean training loss (steps 400-500): 2.502
Mean training loss (steps 500-600): 2.456
Epoch 0 step 672 validation loss: tensor(3.9109, device='cuda:0')
Mean training loss (steps 600-700): 2.375
Mean training loss (steps 700-800): 2.410
Mean training loss (steps 800-900): 2.450
Mean training loss (steps 900-1000): 2.311
Epoch 0 step 1008 validation loss: tensor(3.8125, device='cuda:0')
```

```
Epoch 2 step 9088 validation loss: tensor(3.3974, device='cuda:0')
Mean training loss (steps 2300-2400): 1.776
Mean training loss (steps 2400-2500): 1.749
Mean training loss (steps 2500-2600): 1.786
Epoch 2 step 9424 validation loss: tensor(3.4150, device='cuda:0')
Mean training loss (steps 2600-2700): 1.738
Mean training loss (steps 2700-2800): 1.823
Mean training loss (steps 2800-2900): 1.783
Mean training loss (steps 2900-3000): 1.715
Epoch 2 step 9760 validation loss: tensor(3.4289, device='cuda:0')
Mean training loss (steps 3000-3100): 1.747
Mean training loss (steps 3100-3200): 1.731
Mean training loss (steps 3200-3300): 1.691
Epoch 2 step 10096 validation loss: tensor(3.4246, device='cuda:0')
```

Text generation

```
1   starter_sequence = text_to_id_vector("<sos> This movie is")
2   generated_tokens = language_model.generate(starter_sequence,
3                                              output_length = 30,
4                                              temperature=0.5)
5   generated_text = id_vector_to_text(generated_tokens['output_ids'])
6   print(generated_text)
```

```
made to be a good time <eos> <eos> ) and a overly measured . <eos> a lot of a serious , a lot of the
```

Text evaluation

```
1  with torch.no_grad():
2      movie_text = "<sos> This movie is really quite good! <eos>"
3      movie_sequence = text_to_id_vector(movie_text)
4      movie_likelihood = language_model.evaluate(movie_sequence)
5      print(f'Log-likelihood of "{movie_text}":\n', movie_likelihood)
6  
7      nonmovie_text = "<sos> I ate a sandwich for breakfast. <eos>"
8      nonmovie_sequence = text_to_id_vector(nonmovie_text)
9      nonmovie_likelihood = language_model.evaluate(nonmovie_sequence)
10     print(f'Log-likelihood of "{nonmovie_text}":\n', nonmovie_likelihood)
11 
```

```
Log-likelihood of "<sos> This movie is really quite good! <eos>":
 {'negative_log_likelihood': tensor(11.2020)}
Log-likelihood of "<sos> I ate a sandwich for breakfast. <eos>":
 {'negative_log_likelihood': tensor(13.7354)}
```

TowardsDataScience Tutorial

Lots of ways to do language modeling in Python

e.g. https://towardsdatascience.com/language-modeling-with-lstms-in-pytorch-381a26badcbf

7	2	3	8	12	98	9	87	09	12	23	67	56	45	98	09	48	97	56	11	54	43	76	96
73	26	75	987	87	756	876	87	87	564	658	8	98	92	56	47	97	64	75	87	22	66	289	12
8	72	26	33	78	27	122	21	2	7	8	6	55	42	57	26	72	10	28	23	78	98	65	45
67	45	87	90	8	76	54	3	45	67	65	45	7	62	12	18	72	23	36	34	72	11	33	15

Concluding thoughts

Language modeling kind of complicated in terms of coding

- Much less standardized than classification

Details matter